# *Assembly* do IA-32 em ambiente Linux

### TPC6 e Guião laboratorial

*Alberto José Proença*

---

## Objectivo e notas

A lista de exercícios/tarefas propostos no TPC6 / Guião laboratorial analisa o **suporte a estruturas de controlo e a funções em C**, no IA-32, com recurso a um depurador (*debugger*). Os exercícios para serem resolvidos e entregues antes da aula TP estão assinalados com uma caixa cinza, e repetem-se na última folha. Recomenda-se o uso do mesmo servidor que foi usado na sessão laboratorial anterior, para se garantir coerência na análise e discussão dos resultados.

O texto de "**Introdução ao GDB *debugger***", no fim deste guião, contém informação pertinente ao funcionamento desta sessão laboratorial, e é uma sinopse ultra-compacta do manual; a versão integral está disponível no site da GNU, e recomenda-se ainda a consulta dos documentos disponibilizados nas notas de apoio da disciplina (na Web), por se referirem a versões mais compatíveis com as ferramentas instaladas no servidor.

---

## Ciclo *While*

**1.** Coloque a seguinte função em C num ficheiro com o nome `while_loop.c` , e execute apenas a sua compilação para *assembly*, usando o comando `gcc –O2 –S while_loop.c` .

```
1 int while_loop(int x, int y, int n)
2 {
3   while ((n > 0) & (y < n)) { /* Repare no uso do operador '&' */
4     x += n;
5     y *= n;
6     n--;
7   }
8   return x;
9 }
```

**a)** (A) Considerando que os argumentos `x, y,` e `n,` passados para a função, se encontram respectivamente à distância 8, 12 e 16 do endereço especificado em `%ebp`, **preencha a tabela de utilização de registos** (semelhante ao exemplo da série Fibonacci); considere também a utilização de registos para variáveis temporárias (não visíveis no código C).

| Registo | Variável | Atribuição inicial |
|---|---|---|
| | x | |
| | n | |
| | y | |
| | | |
| | | |
| | | |

**b)** **(R)** Faça nova compilação, usando um comando semelhante mas sem qualquer optimização e criando um ficheiro em *assembly* com uma designação diferente. **Analise os 2 códigos em *assembly* e identifique e caracterize** as principais diferenças que encontrar.

**c)** **Confirme esta utilização dos registos**, directamente no computador:

**i.** **(A)** Construa em C um programa simples (`main`) que use a função `while_loop`, e que não faça mais do que inicializar numericamente um conjunto de valores que irá depois passar como argumento para a função (experimente 4, 2 e 3, respectivamente).
(*Sugestão: use variáveis com designações diferentes das usadas na função*)

**ii.** **(A)** Complete o ficheiro `while_loop.c` com o programa `main` que elaborou, e crie um executável usando o comando `gcc -Wall –O2 -g .`

**iii.** **(A)** Com o comando `objdump –d`, analise o código *assembly* e **identifique** em `while_loop`, **a 1ª instrução** (e respectiva **localização**) **logo a seguir** a: **(i)** leitura de cada um dos argumentos da *stack* (Nota: se o código gerado pelo compilador efectuar esta leitura em 3 instruções consecutivas, basta então identificar apenas a instrução que se segue à última leitura) e **(ii)** utilização pela 1ª vez de cada um dos registos de 8 bits; escreva aqui essas instruções em *assembly* e respetiva localização:

**iv.** **(A)** Invocando o *debugger* (com `gdb <nome_fich_executável>`), **insira pontos de paragem** (*breakpoints*) imediatamente antes da execução dessas instruções; explicite aqui os comandos usados (e registe o nº de *breakpoint* atribuído a cada endereço):

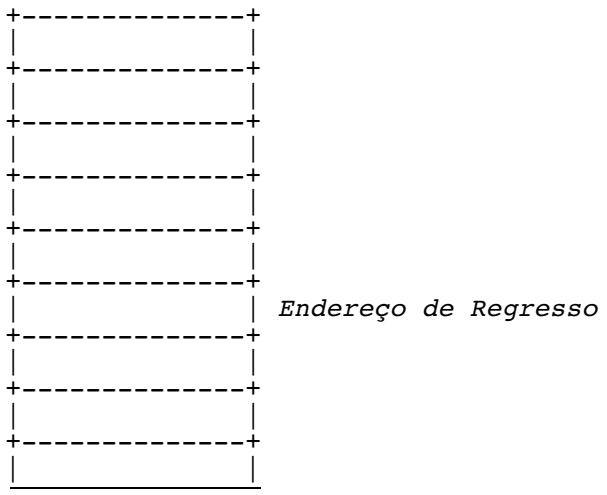
**v.** **(A)** Valide o conjunto de atribuições aos registos, **preenchendo esta tabela sem executar qualquer código** (apenas com base na análise do código).
Depois, **confirme esses valores** executando o programa dentro do *debugger* e, após cada paragem num *breakpoint*, visualizando o conteúdo dos registos (com `print $reg`, ou com `info registers`; de notar que o `gdb` apenas aceita especificação de registos de 32 bits).

| Registo | Variável | Break1 | Break2 | Break3 | Break4 | Break5 |
|---|---|---|---|---|---|---|
| | x | | | | | |
| | n | | | | | |
| | y | | | | | |
| | | | | | | |
| | | | | | | |

**d)** **(R/B)** Com base nos argumentos passados para a função `while_loop` (no `main`), é possível estimar quantas vezes o *loop* é executado na função. Para confirmar esse valor, uma técnica é introduzir um *breakpoint* na instrução de salto condicional de regresso ao início do *loop*. Indique o que deve fazer depois para confirmar esse valor.

**e)** **(A/R)** Considerando que a *stack* cresce para cima, pretende-se construir o diagrama da *stack frame* da função `while_loop` com o máx. de indicações (endereços e conteúdos; pode ser em hexadecimal), logo após a execução da instrução antes do 2º *breakpoint*. Comente cada um dos conteúdos da *stack frame* (por ex., "endereço de regresso").

**Construa assim esse diagrama**: **(i)** estimando os valores antes da execução do código, e **(ii)** confirmando posteriormente esses valores, usando o depurador durante a execução do código.




**f)** **(A/R)** Identifique a expressão de teste e o corpo do ciclo `while` (*body-statement*) no bloco do código C, e assinale as linhas de código no programa em *assembly* que lhe são correspondentes. Que otimizações foram feitas pelo compilador?

**g)** **(R)** Escreva uma versão do tipo `goto` (em C) da função, com uma estrutura semelhante ao do código *assembly* (tal como foi feito para a série Fibonacci).
(*Para fazer depois da sessão laboratorial*)

# Anexo: Introdução ao GNU *debugger*

O GNU *debugger* GDB disponibiliza um conjunto de funcionalidades úteis na análise e avaliação do funcionamento de programas em linguagem máquina, durante a sua execução; permite ainda a execução controlada de um programa, com indicação explícita de quando interromper essa execução – através de *breakpoints*, ou em execução passo-a-passo – e possibilitando a análise do conteúdo de registos e de posições de memória, após cada interrupção.
Use o GDB para confirmar as tabelas de utilização de registos e o valor dos argumentos nos exercícios. Nota: utilize primeiro `objdump` para obter uma versão "desmontada" do programa.

A tabela/figura seguinte (de CSAPP) ilustra a utilização de alguns dos comandos mais comuns para o IA32.

| Command | Effect |
|---|---|
| **Starting and Stopping** | |
| *quit* | Exit GDB |
| *run* | Run your program (give command line argum. here) |
| *kill* | Stop your program |
| **Breakpoints** | |
| *break sum* | Set breakpoint at entry to function `sum` |
| *break *0x80483c3* | Set breakpoint at address `0x80483c3` |
| *disable* 3 | Disable breakpoint 3 |
| *enable* 2 | Enable breakpoint 2 |
| *clear* sum | Clear any breakpoint at entry to function `sum` |
| *delete* 1 | Delete breakpoint 1 |
| *delete* | Delete all breakpoints |
| **Execution** | |
| *stepi* | Execute one instruction |
| *stepi 4* | Execute four instructions |
| *nexti* | Like `stepi`, but proceed through function calls |
| *continue* | Resume execution |
| *finish* | Run until current function returns |
| **Examining code** | |
| *disas* | Disassemble current function |
| *disas sum* | Disassemble function sum |
| *disas 0x80483b7* | Disassemble function around address `0x80483b7` |
| *disas 0x80483b7 0x80483c7* | Disassemble code within specified address range |
| *print /x $eip* | Print program counter in hex |
| **Examining data** | |
| *print $eax* | Print contents of `%eax` in decimal |
| *print /x $eax* | Print contents of `%eax` in hex |
| *print /t $eax* | Print contents of `%eax` in binary |
| *print 0x100* | Print decimal representation of `0x100` |
| *print /x 555* | Print hex representation of 555 |
| *print /x ($ebp+8)* | Print contents of `%ebp` plus 8 in hex |
| *print *(int *) 0xbffff890* | Print integer at address `0xbffff890` |
| *print *(int *) ($ebp+8)* | Print integer at address `%ebp` + 8 |
| *x/2w 0xbffff890* | Examine 2(4-byte) words starting at addr 0xbffff890 |
| *x/20b sum* | Examine first 20 bytes of function `sum` |
| **Useful information** | |
| *info frame* | Information about current stack frame |
| *info registers* | Values of all the registers |
| *help* | Get information about GDB |

Figure 3.27: **Example GDB Commands.** These examples illustrate some of the ways GDB supports debugging of machine-level programs.

| Nº | Nome: | Turma: |
|---|---|---|

## Resolução dos exercícios

### 1. (A) Análise do código em *assembly*

```
1 int while_loop(int x, int y, int n)
2 {
3   while ((n > 0) & (y < n)) { /* Repare no uso do operador '&' */
4     x += n;
5     y *= n;
6     n--;
7   }
8   return x;
9 }
```

Código otimizado em *assembly*:

| Registo | Variável | Atribuição inicial |
|---|---|---|
| | x | |
| | n | |
| | y | |
| | | |
| | | |
| | | |

Principais diferenças entre as versões do *assembly* (otimizada e não-otimizada):

Código C de um programa simples (`main`) que usa a função `while_loop` :